# CS 152 Computer Architecture and Engineering

# Lecture 2 - Simple Machine Implementations, Microcode

Dr. George Michelogiannakis
EECS, University of California at Berkeley
CRD, Lawrence Berkeley National Laboratory

**`http://inst.eecs.berkeley.edu/~cs152`**

# Last Time in Lecture 1

- Computer Architecture >> ISAs and RTL
  - CS152 is about interaction of hardware and software, and design of appropriate abstraction layers
- The end of the uniprocessor era
  - With simple and specialized cores due to power constraints
- Cost of software development becomes a large constraint on architecture (need compatibility)
- IBM 360 introduces notion of “family of machines” running same ISA but very different implementations
  - Six different machines released on same day (April 7, 1964)
  - “Future-proofing” for subsequent generations of machine

# Question of the Day

- What purpose does microcode serve today?
  - Would we have it if designing ISAs from scatch?
  - Why would we want a complex ISA?
  - Why do you think motivated CISC and RISC?

# Instruction Set Architecture (ISA)

- The contract between software and hardware
- Typically described by giving all the programmer-visible state (registers + memory) plus the semantics of the instructions that operate on that state
- IBM 360 was first line of machines to separate ISA from implementation (aka. *microarchitecture*)
- Many implementations possible for a given ISA
  - E.g., the Soviets build code-compatible clones of the IBM360, as did Amdahl after he left IBM.
  - E.g.2., today you can buy AMD or Intel processors that run the x86-64 ISA.
  - E.g.3: many cellphones use the ARM ISA with implementations from many different companies including TI, Qualcomm, Samsung, Marvell, etc.

# Name a Famous ISA!

- Intel’s x86 was initially deployed in 1978
- Is alive and well today, though larger
- Reference manual has 3883 pages!

# Implementations of the x86

- Hundreds of different processors implement x86
  – Not just by Intel




- Some have extensions that compilers can use if available
  – But software still compatible if not
- More than just intel develop x86
  – X86-64 was first specified by AMD in 2000

# ISA to Microarchitecture Mapping

- ISA often designed with particular microarchitectural style in mind, e.g.,
  - Accumulator $\Rightarrow$ hardwired, unpipelined
  - CISC $\Rightarrow$ microcoded
  - RISC $\Rightarrow$ hardwired, pipelined
  - VLIW $\Rightarrow$ fixed-latency in-order parallel pipelines
  - JVM $\Rightarrow$ software interpretation
- But can be implemented with any microarchitectural style
  - Intel Ivy Bridge: hardwired pipelined CISC (x86) machine (with some microcode support)
  - Simics: Software-interpreted SPARC RISC machine
  - ARM Jazelle: A hardware JVM processor
  - This lecture: a microcoded RISC-V machine

# Today, Microprogramming

- To show how to build very small processors with complex ISAs
- To help you understand where CISC* machines came from
- Because still used in common machines (IBM360, x86, PowerPC)
- As a gentle introduction into machine structures
- To help understand how technology drove the move to RISC*

* “CISC”/”RISC” names much newer than style of machines they refer to.

# Problem Microprogramming Solves

- Complex ISA to ease programmer and assembler’s life
  - With instructions that have multiple steps

- Simple processors such as in order to meet power constraints
  - (refer to previous lecture)
- Turn complex architecture into simple microarchitecture with programmable control
- Can also patch microcode

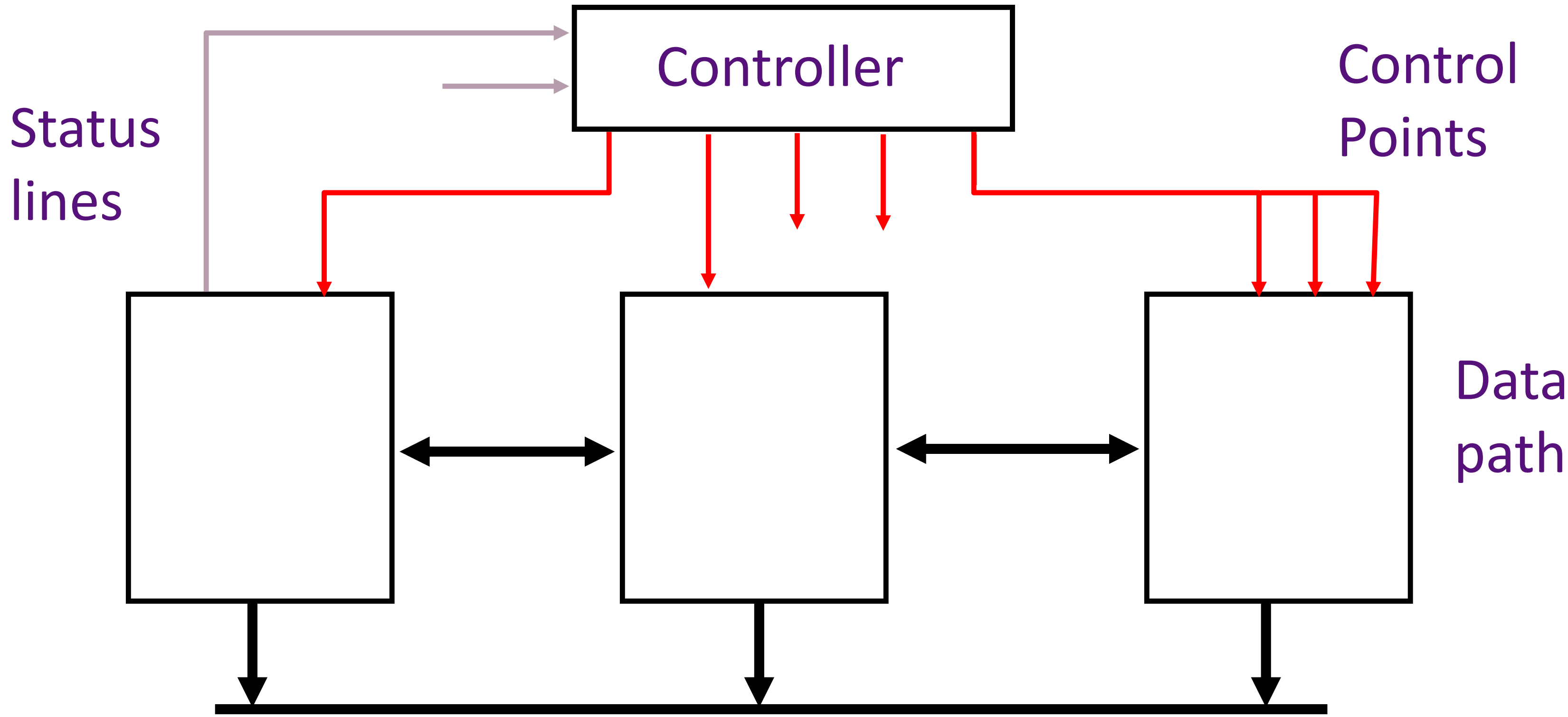
# The Idea

- An ISA (assembly) instruction, is not what drives the processor's datapath directly

- Instead, instructions are broken down to FSM states

- Each state is a microinstruction and outputs control signals

# Microarchitecture: Bus-Based *Implementation of ISA*




*Structure:* How components are connected.

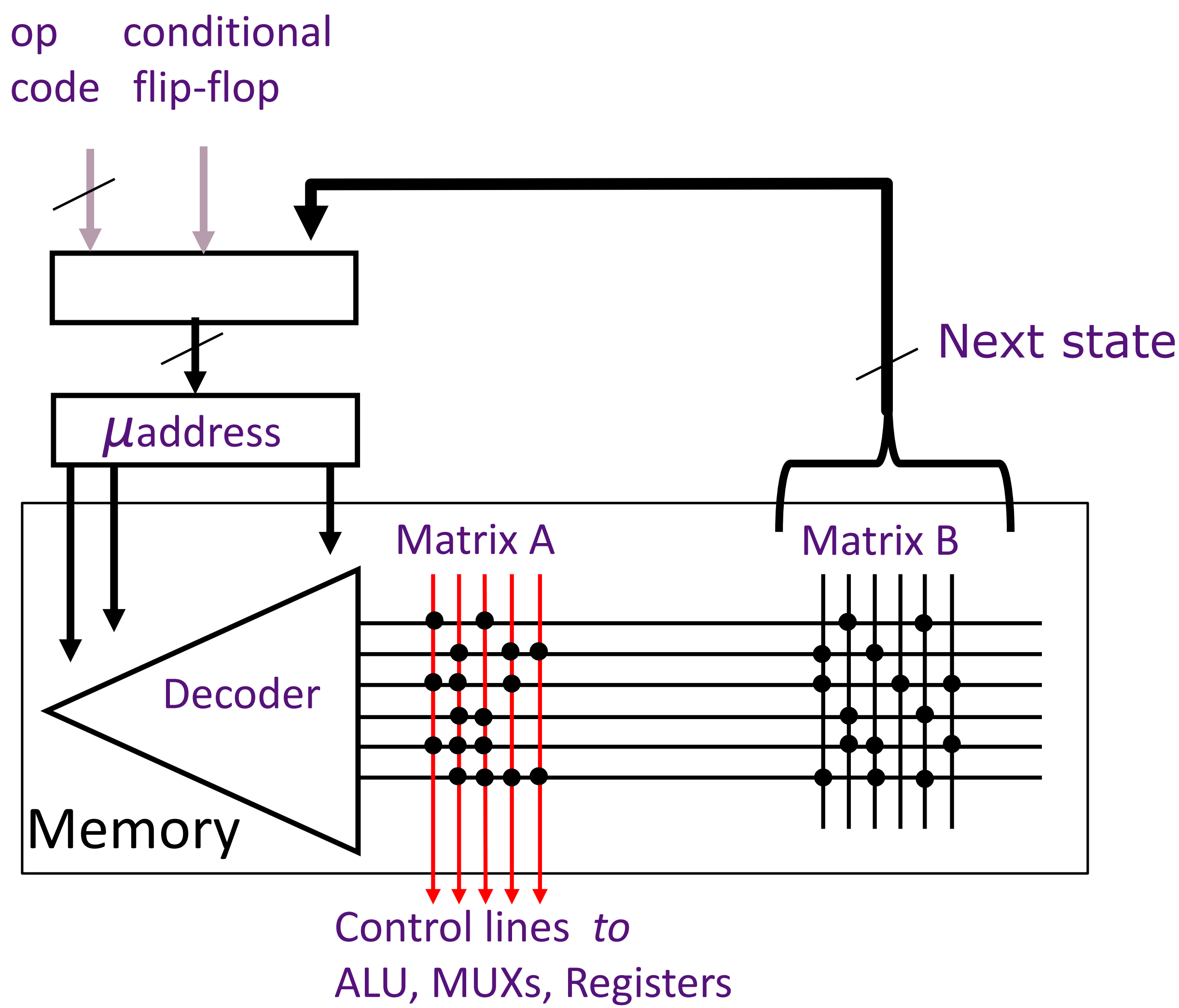
*Static*

*Behavior:* How data moves between components

*Dynamic*

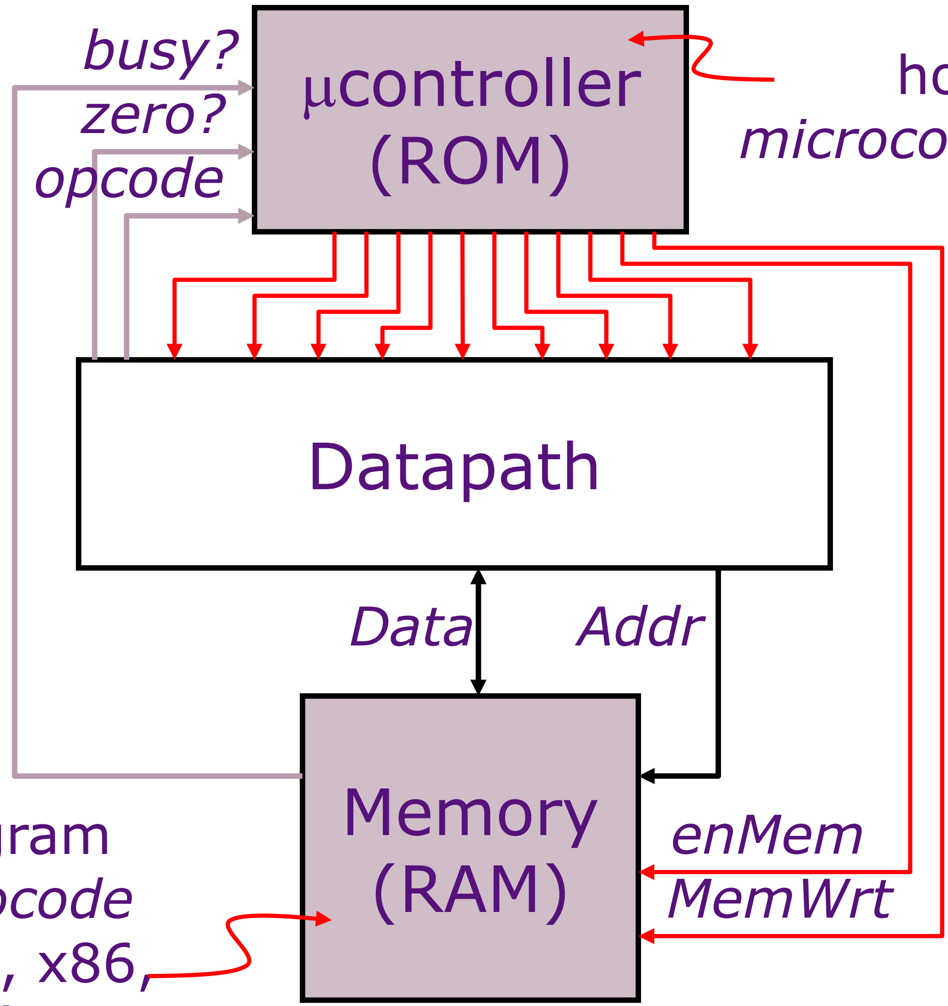
# Microcontrol Unit *Maurice Wilkes, 1954*

*First used in EDSAC-2, completed 1958*




*Embed the control logic state table in a memory array*

# Microcoded Microarchitecture




holds fixed *microcode* instructions

holds user program written in *macrocode* instructions (e.g., x86, RISC-V, etc.)

# RISC-V ISA

- RISC design from UC Berkeley
- Realistic & complete ISA, but open & simple
- Not over-architected for a certain implementation style
- Both 32-bit and 64-bit address space variants
  - RV32 and RV64
- Easy to subset/extend for education/research
  - RV32IM, RV32IMA, RV32IMAFD, RV32G
- Techreport with RISC-V spec available on class website or riscv.org
- We’ll be using 32-bit and 64-bit RISC-V this semester in lectures and labs. Similar to MIPS you saw in CS61C

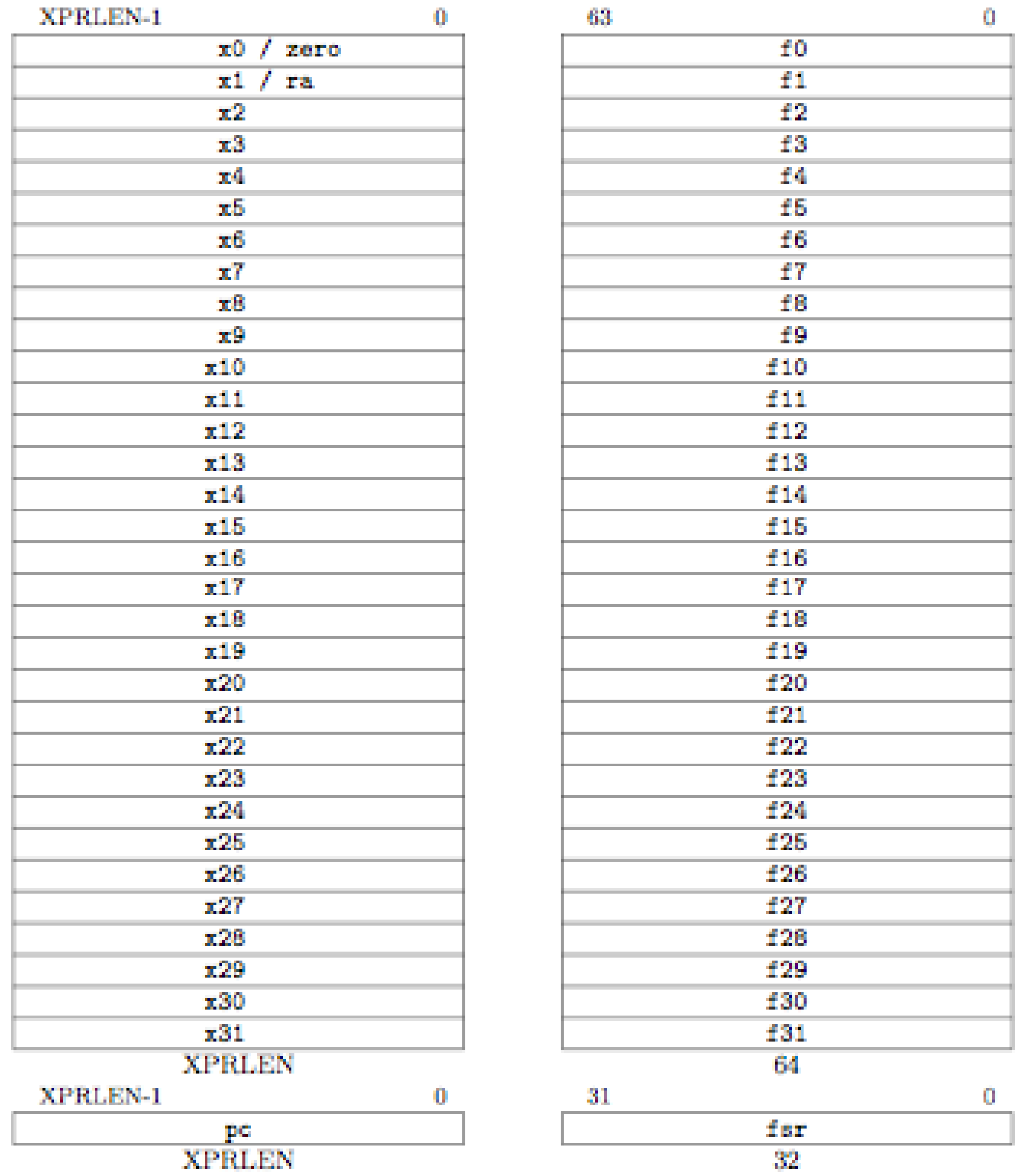
# RV32 Processor State

Program counter (**pc**)

32x32-bit integer registers (**x0-x31**)
- **x0** always contains a 0

32 floating-point (FP) registers (**f0-f31**)
- each can contain a single- or double-precision FP value (32-bit or 64-bit IEEE FP)
- Is an extension

FP status register (**fsr**), used for FP rounding mode & exception reporting

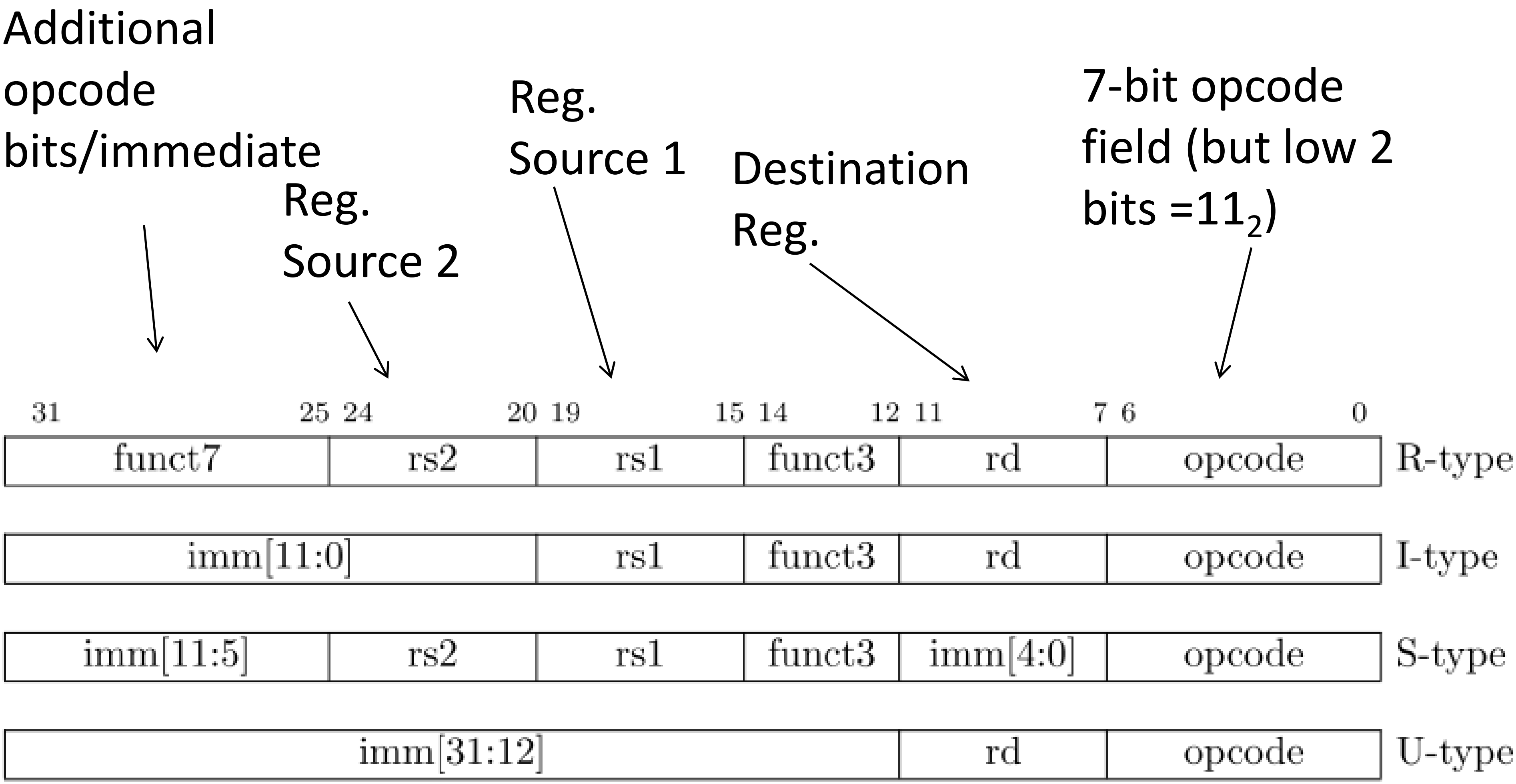


# RISC-V Instruction Encoding

| | | | |
|---|---|---|---|
| | | xxxxxxxxxxxxxxaa | 16-bit (aa ≠ 11) |
| | xxxxxxxxxxxxxxxx | xxxxxxxxxxxbbb11 | 32-bit (bbb ≠ 111) |
| ···xxxx | xxxxxxxxxxxxxxxx | xxxxxxxxxx011111 | 48-bit |
| ···xxxx | xxxxxxxxxxxxxxxx | xxxxxxxxx0111111 | 64-bit |
| ···xxxx | xxxxxxxxxxxxxxxx | xxxxxnnnn1111111 | (80+16*nnnn)-bit, nnnn≠1111 |
| ···xxxx | xxxxxxxxxxxxxxxx | xxxxx11111111111 | Reserved for ≥320-bits |

- Base instruction set (RV32) always has fixed 32-bit instructions lowest two bits = $11_2$
- All branches and jumps have targets at 16-bit granularity (even in base ISA where all instructions are fixed 32 bits)
  - Still will cause a fault if fetching a 32-bit instruction

# Four Core RISC-V Instruction Formats




Aligned on a four-byte boundary in memory. There are variants!

Sign bit of immediates always on bit 31 of instruction. Register fields never move

# With Variants

| 31 | 30 25 | 24 21 | 20 | 19 15 | 14 12 | 11 8 | 7 | 6 0 | |
|---|---|---|---|---|---|---|---|---|---|
| funct7 | | rs2 | | rs1 | funct3 | rd | | opcode | R-type |
| imm[11] | imm[10:5] | imm[4:1] | imm[0] | rs1 | funct3 | rd | | opcode | I-type |
| imm[11] | imm[10:5] | rs2 | | rs1 | funct3 | imm[4:1] | imm[0] | opcode | S-type |
| imm[12] | imm[10:5] | rs2 | | rs1 | funct3 | imm[4:1] | imm[11] | opcode | SB-type |
| imm[31] | imm[30:20] | | | imm[19:15] | imm[14:12] | rd | | opcode | U-type |
| imm[20] | imm[10:5] | imm[4:1] | imm[11] | imm[19:15] | imm[14:12] | rd | | opcode | UJ-type |

# Integer Computational Instructions

- I-type
- ADDI: adds sign extended 12-bit immediate to rs1
  - Actually, all immediates in all instructions are sign extended
- SLTI(U): set less than immediate
- Shift instructions, etc…

| 31 … 20 | 19 … 15 | 14 … 12 | 11 … 7 | 6 … 0 |
|---|---|---|---|---|
| imm[11:0] | rs1 | funct3 | rd | opcode |
| 12 | 5 | 3 | 5 | 7 |
| I-immediate[11:0] | src | ADDI/SLTI[U] | dest | OP-IMM |
| I-immediate[11:0] | src | ANDI/ORI/XORI | dest | OP-IMM |

# Integer Computational Instructions

- R-type
- Rs1 and rs2 are the source registers. Rd the destination
- SLT, SLTU: set less than
- SRL, SLL, SRA: shift logical or arithmetic left or right

| 31 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|
| funct7 | rs2 | rs1 | funct3 | rd | opcode |
| 7 | 5 | 5 | 3 | 5 | 7 |
| 0000000 | src2 | src1 | ADD/SLT/SLTU | dest | OP |
| 0000000 | src2 | src1 | AND/OR/XOR | dest | OP |
| 0000000 | src2 | src1 | SLL/SRL | dest | OP |
| 0100000 | src2 | src1 | SUB/SRA | dest | OP |

# S-Type

12-bit signed immediate split across two fields

| 31 | 30 — 25 | 24 — 20 | 19 — 15 | 14 — 12 | 11 — 8 | 7 | 6 — 0 |
|---|---|---|---|---|---|---|---|
| imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | opcode |
| 1 | 6 | 5 | 5 | 3 | 4 | 1 | 7 |
| offset[12,10:5] | | src2 | src1 | BEQ/BNE | offset[11,4:1] | | BRANCH |
| offset[12,10:5] | | src2 | src1 | BLT[U] | offset[11,4:1] | | BRANCH |
| offset[12,10:5] | | src2 | src1 | BGE[U] | offset[11,4:1] | | BRANCH |

Branches, compare two registers, PC+(immediate<<1) target

*(Signed offset in multiples of two).* Branches do not have delay slot

# UJ-Type

| 31 | 30 — 21 | 20 | 19 — 12 | 11 — 7 | 6 — 0 |
|---|---|---|---|---|---|
| imm[20] | imm[10:1] | imm[11] | imm[19:12] | rd | opcode |
| 1 | 10 | 1 | 8 | 5 | 7 |
| offset[20:1] | | | | dest | JAL |

"J" Unconditional jump, PC+offset target

"JAL" Jump and link, also writes PC+4 to **x1**

Offset scaled by 1-bit left shift – can jump to 16-bit instruction boundary (Same for branches)

Also "JALR" where Imm (12 bits) + rd1 = target

# L-Type

| 31 — 12 | 11 — 7 | 6 — 0 |
|---|---|---|
| imm[31:12] | rd | opcode |
| 20 | 5 | 7 |
| U-immediate[31:12] | dest | LUI |
| U-immediate[31:12] | dest | AUIPC |

Writes 20-bit immediate to top of destination register.

Used to build large immediates.

12-bit immediates are signed, so have to account for sign when building 32-bit immediates in 2-instruction sequence (LUI high-20b, ADDI low-12b)

# Loads and Stores

| 31 — 20 | 19 — 15 | 14 — 12 | 11 — 7 | 6 — 0 |
|---|---|---|---|---|
| imm[11:0] | rs1 | funct3 | rd | opcode |
| 12 | 5 | 3 | 5 | 7 |
| offset[11:0] | base | width | dest | LOAD |

| 31 — 25 | 24 — 20 | 19 — 15 | 14 — 12 | 11 — 7 | 6 — 0 |
|---|---|---|---|---|---|
| imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode |
| 7 | 5 | 5 | 3 | 5 | 7 |
| offset[11:5] | src | base | width | offset[4:0] | STORE |

Store instructions (S-type). Loads (I-type).

(rs1 + immediate) addressing

Store only uses rs1 and rs2. Rd is only present when being written to

# Where is NOP?

addi x0, x0, 0

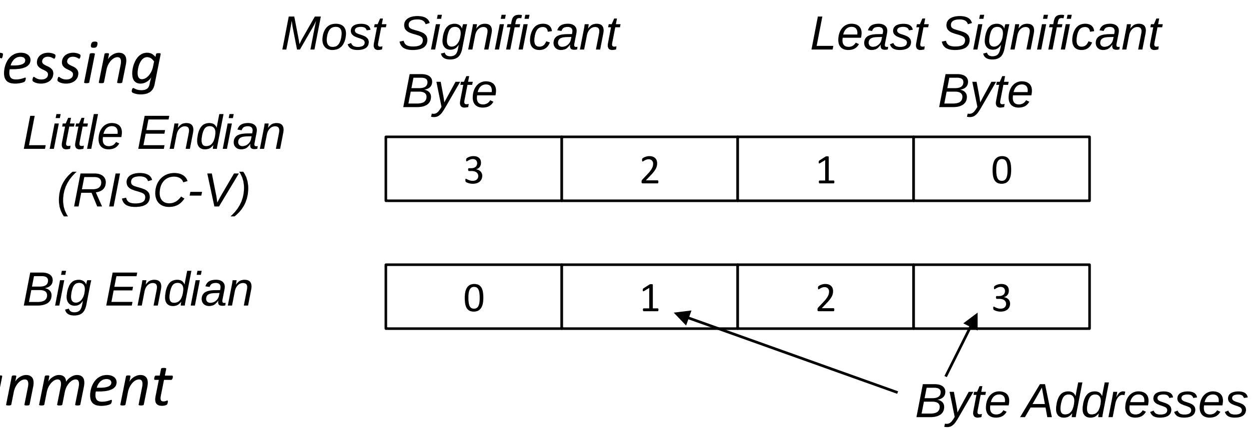
# Data Formats and Memory Addresses

Data formats:

8-b Bytes, 16-b Half words, 32-b words and 64-b double words

Some issues

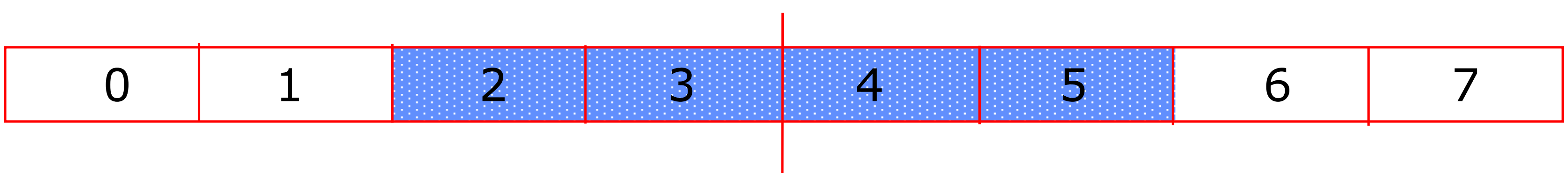
- *Byte addressing*




- *Word alignment*

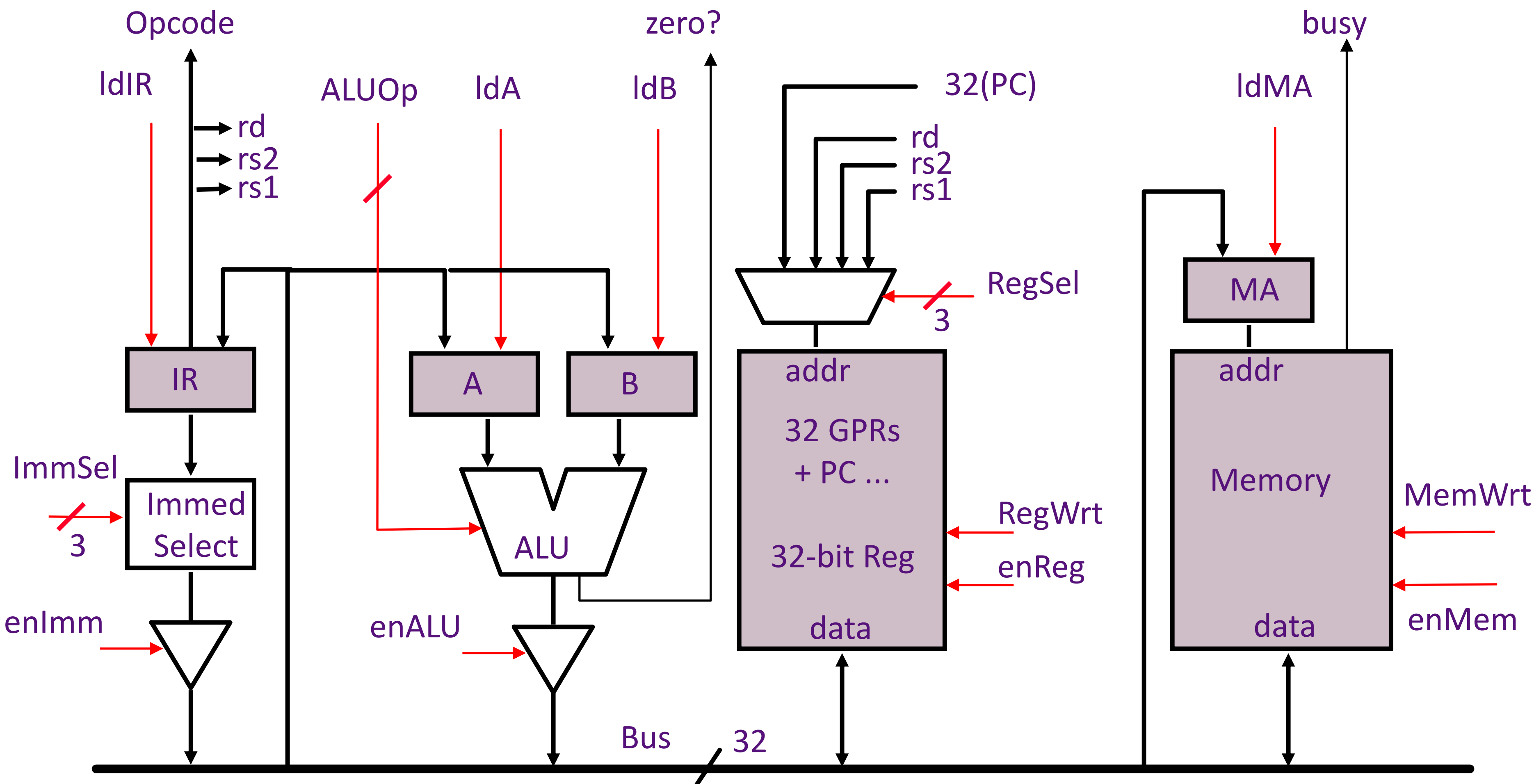
  Suppose the memory is organized in 32-bit words.

  Can a word address begin only at 0, 4, 8, .... ?



# BACK TO MICROCODING

# A Bus-based Datapath for RISC-V

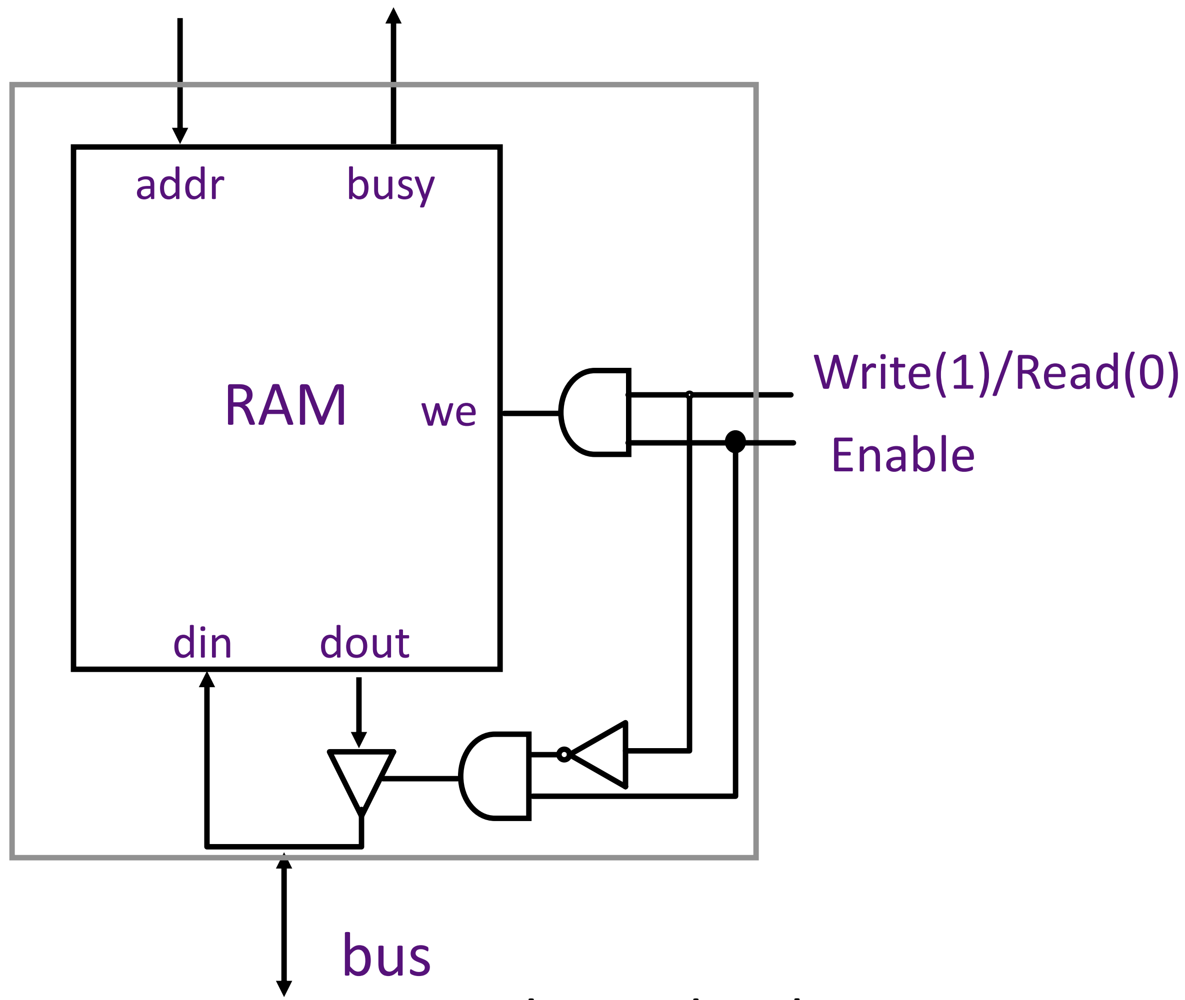



*Microinstruction: register to register transfer (17 control signals)*

MA <= PC *means* RegSel = PC; enReg=yes; ldMA= yes

B <= Reg[rs2] *means* RegSel = rs2; enReg=yes; ldB = yes

# Memory Module




Assumption: Memory operates independently and is slow as compared to Reg-to-Reg transfers (multiple CPU clock cycles per access)

# Instruction Execution

Execution of a RISC-V instruction involves:

1. instruction fetch
2. decode and register fetch
3. ALU operation
4. memory operation (optional)
5. write back to register file (optional)

   + the computation of the
   *next instruction* address

# Microprogram Fragments

instr fetch: MA, A <= PC
PC <= A + 4
IR <= Memory
dispatch on Opcode

*can be treated as a macro*

ALU: A <= Reg[rs1]
B <= Reg[rs2]
Reg[rd] <= func(A,B)
*do* instruction fetch

ALUi: A <= Reg[rs1]
B <= Imm *sign extension*
Reg[rd] <= Opcode(A,B)
*do* instruction fetch

# Microprogram Fragments *(cont.)*

LW:

A <= Reg[rs1]
B <= Imm
MA <= A + B
Reg[rd] <= Memory
*do* instruction fetch

J:
(JAL with rd=x0)

A <= A - 4 Get original PC back in A
B <= IR
PC <= JumpTarg(A,B)
*do* instruction fetch

*JumpTarg(A,B) =*
*{A + (B[31:7]<<1)}*

beq:

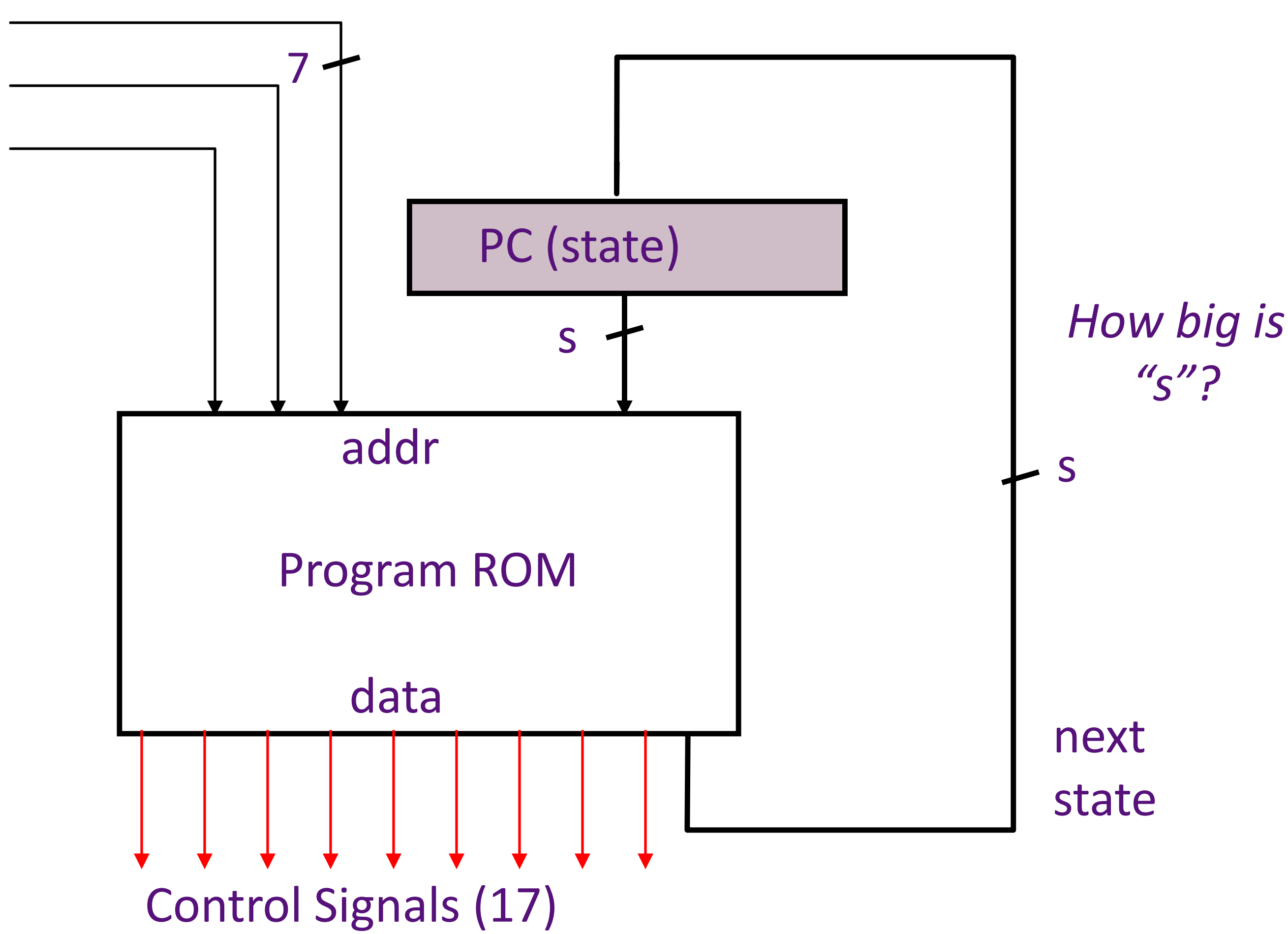
A <= Reg[rs1]
B <= Reg[rs2]
*If* A==B *then go to* bz-taken
*do* instruction fetch

bz-taken:

A <= PC
A <= A - 4 Get original PC back in A
B <= BImm << 1 BImm = IR[31:27,16:10]
PC <= A + B
*do* instruction fetch

# RISC-V Microcontroller: *first attempt*

## *pure ROM implementation*




*ROM size ?*

= $2^{(opcode+sbits)}$ words

*Word size ?*

= control+s bits

# Microprogram in the ROM *worksheet*

| State | Op | zero? | busy | Control points | next-state |
|---|---|---|---|---|---|
| $fetch_0$ | * | * | * | MA,A <= PC | $fetch_1$ |
| $fetch_1$ | * | * | yes | .... | $fetch_1$ |
| $fetch_1$ | * | * | no | IR <= Memory | $fetch_2$ |
| $fetch_2$ | * | * | * | PC <= A + 4 | ? |
| $fetch_2$ | ALU | * | * | PC <= A + 4 | $ALU_0$ |
| $ALU_0$ | * | * | * | A <= Reg[rs1] | $ALU_1$ |
| $ALU_1$ | * | * | * | B <= Reg[rs2] | $ALU_2$ |
| $ALU_2$ | * | * | * | Reg[rd] <= func(A,B) | $fetch_0$ |

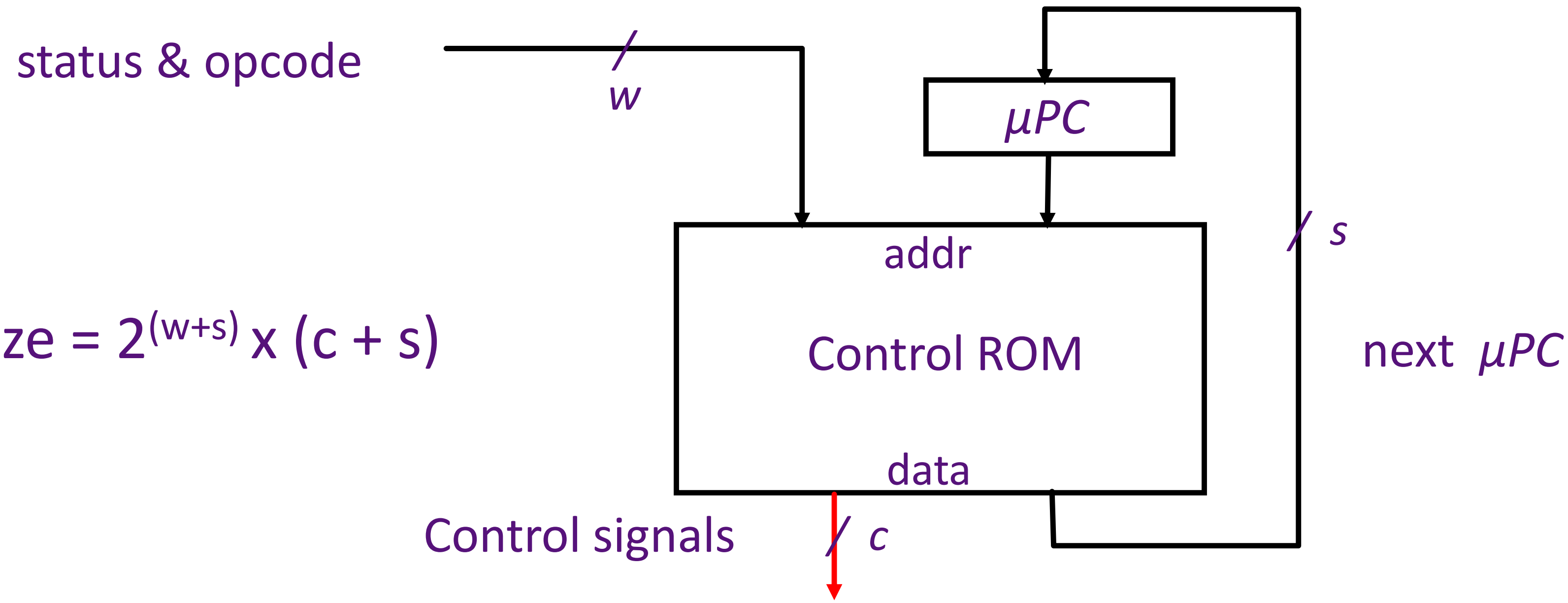
# Microprogram in the ROM

| State | Op | zero? | busy | Control points | next-state |
|---|---|---|---|---|---|
| $fetch_0$ | * | * | * | MA,A <= PC | $fetch_1$ |
| $fetch_1$ | * | * | yes | .... | $fetch_1$ |
| $fetch_1$ | * | * | no | IR <= Memory | $fetch_2$ |
| $fetch_2$ | ALU | * | * | PC <= A + 4 | $ALU_0$ |
| $fetch_2$ | ALUi | * | * | PC <= A + 4 | $ALUi_0$ |
| $fetch_2$ | LW | * | * | PC <= A + 4 | $LW_0$ |
| $fetch_2$ | SW | * | * | PC <= A + 4 | $SW_0$ |
| $fetch_2$ | J | * | * | PC <= A + 4 | $J_0$ |
| $fetch_2$ | JAL | * | * | PC <= A + 4 | $JAL_0$ |
| $fetch_2$ | JR | * | * | PC <= A + 4 | $JR_0$ |
| $fetch_2$ | JALR | * | * | PC <= A + 4 | $JALR_0$ |
| $fetch_2$ | beq | * | * | PC <= A + 4 | $beq_0$ |
| ... | | | | | |
| $ALU_0$ | * | * | * | A <= Reg[rs1] | $ALU_1$ |
| $ALU_1$ | * | * | * | B <= Reg[rs2] | $ALU_2$ |
| $ALU_2$ | * | * | * | Reg[rd] <= func(A,B) | $fetch_0$ |

# Microprogram in the ROM *Cont.*

| State | Op | zero? | busy | Control points | next-state |
|---|---|---|---|---|---|
| $ALUi_0$ | * | * | * | A <= Reg[rs1] | $ALUi_1$ |
| $ALUi_1$ | * | * | * | B <= Imm | $ALUi_2$ |
| $ALUi_2$ | * | * | * | Reg[rd]<= Op(A,B) | $fetch_0$ |
| ... | | | | | |
| $J_0$ | * | * | * | A <= A - 4 | $J_1$ |
| $J_1$ | * | * | * | B <= IR | $J_2$ |
| $J_2$ | * | * | * | PC <= JumpTarg(A,B) | $fetch_0$ |
| ... | | | | | |
| $beq_0$ | * | * | * | A <= Reg[rs1] | $beq_1$ |
| $beq_1$ | * | * | * | B <= Reg[rs2] | $beq_2$ |
| $beq_2$ | * | yes | * | A <= PC | $beq_3$ |
| $beq_2$ | * | no | * | .... | $fetch_0$ |
| $beq_3$ | * | * | * | A <= A - 4 | $beq_4$ |
| $beq_4$ | * | * | * | B <= BImm | $beq_5$ |
| $beq_5$ | * | * | * | PC <= A+B | $fetch_0$ |
| ... | | | | | |

# Size of Control Store




size = $2^{(w+s)} \times (c + s)$

*RISC-V:* w = 5+2 c = 17 s = ?

no. of steps per opcode = 4 to 6 + fetch-sequence

no. of states ~= (4 steps per op-group ) x op-groups + common sequences

= 4 x 8 + 10 states = 42 states => s = 6

Control ROM = $2^{(5+6)}$ x 23 bits approx. 24 Kbytes

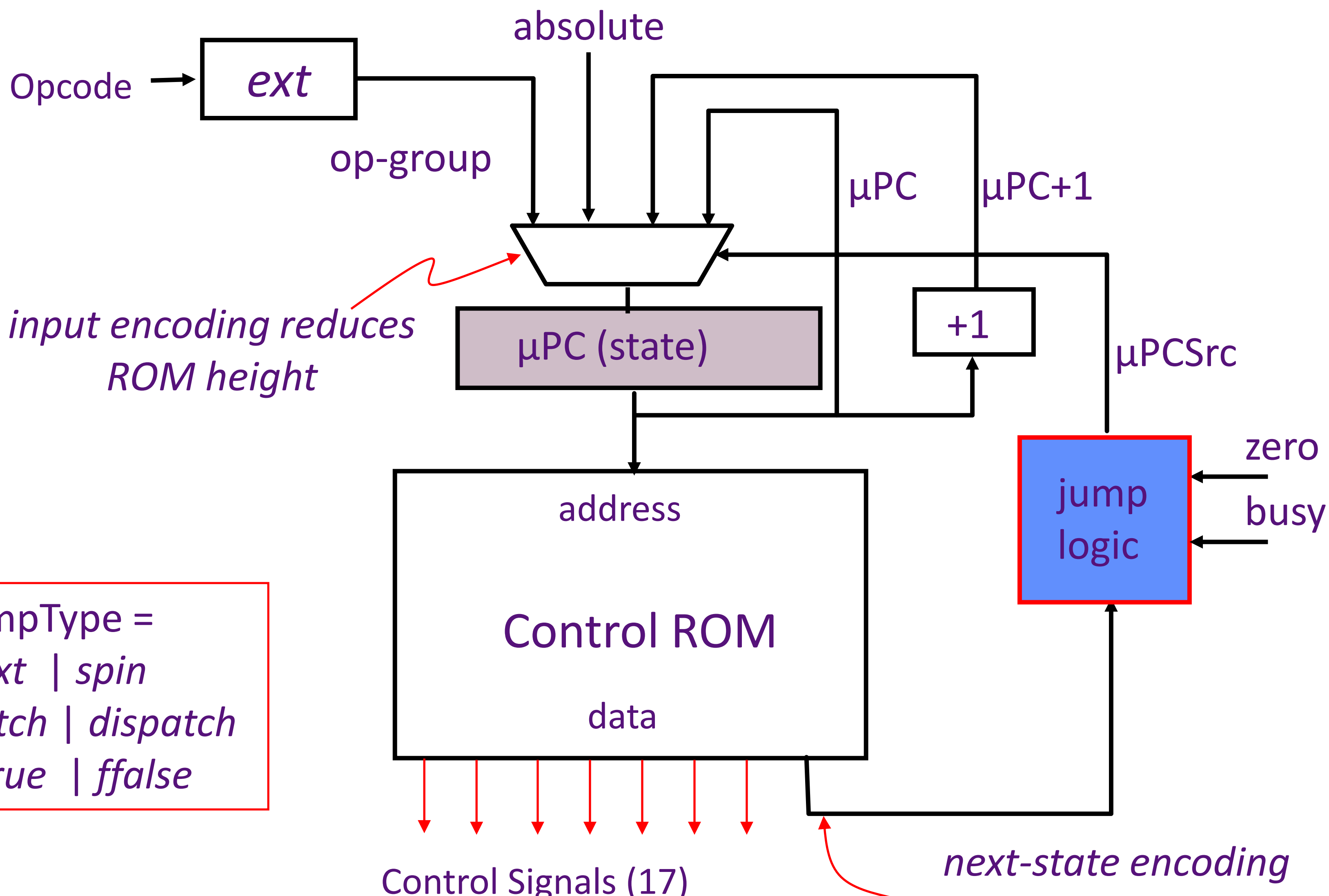
# Reducing Control Store Size

Control store has to be *fast* => *expensive*

- Reduce the ROM height (= address bits)
  - *reduce inputs by extra external logic*
    each input bit doubles the size of the control store
  - *reduce states by grouping opcodes*
    find common sequences of actions
  - *condense input status bits*
    combine all exceptions into one, i.e., exception/no-exception

- Reduce the ROM width
  - *restrict the next-state encoding*
    Next, Dispatch on opcode, Wait for memory, ...
  - *encode control signals (vertical microcode)*

# RISC-V Controller V2




µJumpType =
*next* | *spin*
| *fetch* | *dispatch*
| *ftrue* | *ffalse*

# Jump Logic

μPCSrc = *Case* μJumpTypes

next=> μPC+1

spin => if (busy) then μPC else μPC+1

fetch => absolute

dispatch => op-group

ftrue => if (zero) then absolute else μPC+1

ffalse => if (zero) then μPC+1 else absolute

# Instruction Fetch & ALU: *RISC-V-Controller-2*

| State | Control points | next-state |
| --- | --- | --- |
| $fetch_0$ | MA,A <= PC | next |
| $fetch_1$ | IR <= Memory | spin |
| $fetch_2$ | PC <= A + 4 | dispatch |
| ... | | |
| $ALU_0$ | A <= Reg[rs1] | next |
| $ALU_1$ | B <= Reg[rs2] | next |
| $ALU_2$ | Reg[rd]<=func(A,B) | fetch |
| | | |
| $ALUi_0$ | A <= Reg[rs1] | next |
| $ALUi_1$ | B <= Imm | next |
| $ALUi_2$ | Reg[rd]<= Op(A,B) | fetch |

# Load & Store: *RISC-V-Controller-2*

| State | Control points | next-state |
|---|---|---|
| $LW_0$ | A <= Reg[rs1] | next |
| $LW_1$ | B <= Imm | next |
| $LW_2$ | MA <= A+B | next |
| $LW_3$ | Reg[rd] <= Memory | spin |
| $LW_4$ | | fetch |
| | | |
| $SW_0$ | A <= Reg[rs1] | next |
| $SW_1$ | B <= BImm | next |
| $SW_2$ | MA <= A+B | next |
| $SW_3$ | Memory <= Reg[rs2] | spin |
| $SW_4$ | | fetch |

# Branches: *RISC-V-Controller-2*

| State | Control points | next-state |
|---|---|---|
| $beq_0$ | A <= Reg[rs1] | next |
| $beq_1$ | B <= Reg[rs2] | next |
| $beq_2$ | A <= PC | ffalse |
| $beq_3$ | A <= A- 4 | next |
| $beq_3$ | B <= BImm<<1 | next |
| $beq_4$ | PC <= A+B | fetch |

# Jumps: *RISC-V-Controller-2*

| State | Control points | next-state |
|---|---|---|
| $JALR_0$ | A <= Reg[rs1] | next |
| $JALR_1$ | Reg[1] <= A | next |
| $JALR_2$ | PC <= A | fetch |
| $JAL_0$ | A <= PC | next |
| $JAL_1$ | Reg[1] <= A | next |
| $JAL_2$ | A <= A-4 | next |
| $JAL_3$ | B <= IR | next |
| $JAL_4$ | PC <= JumpTarg(A,B) | fetch |

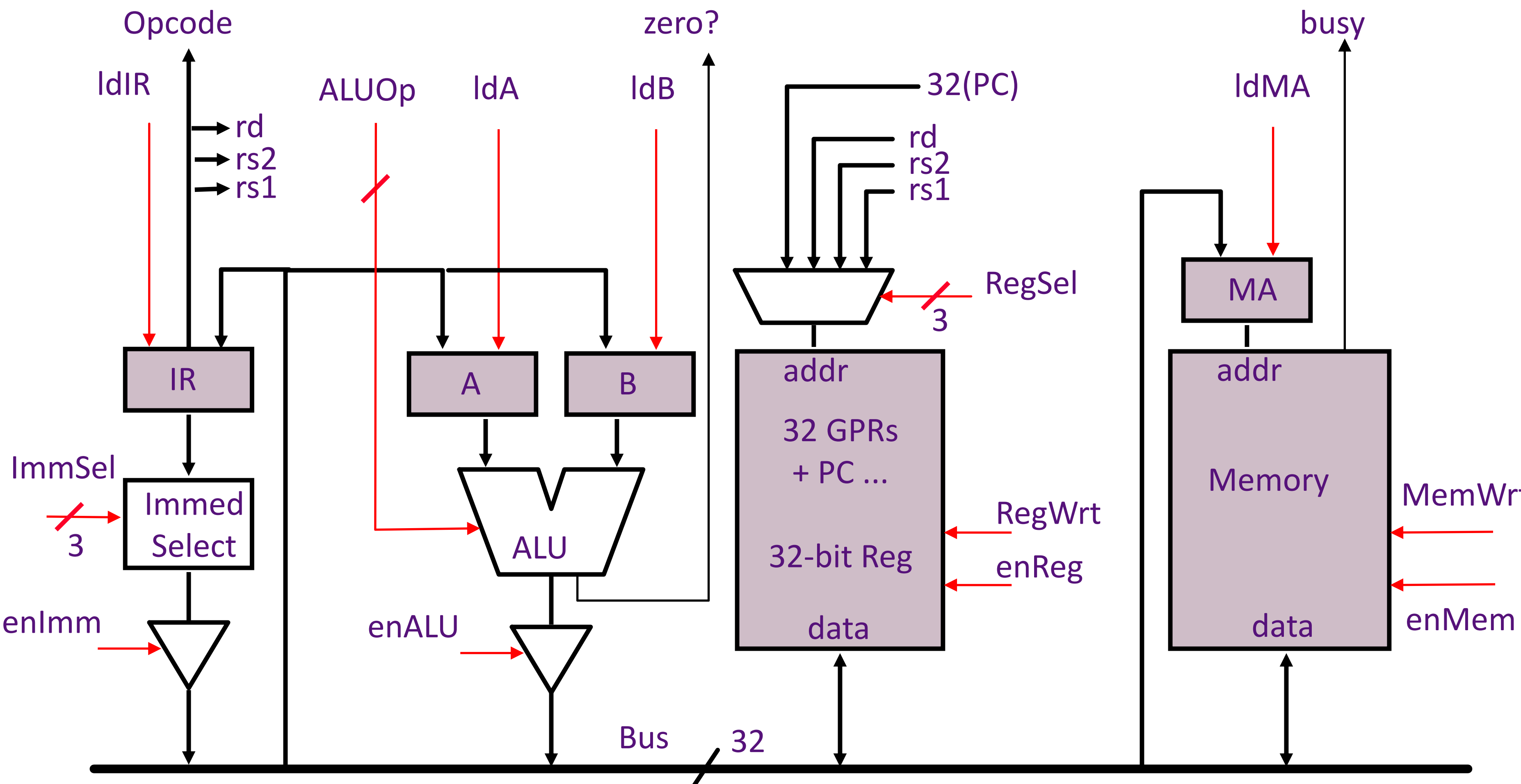
J and JR are special cases with rd = x0

# VAX 11-780 Microcode

```
; P1WFUD.!   )[600,1205]    MICRO2  1F(12)    26-May-81  14:58:1        VAX11/780 Microcode : PCS 01, FPLA 0D, WCS122       'age  771
; CALL2 .MIC  [600,1205]    Procedure call      : CALLG, CALLS

                                                ;29744  ;HERE FOR CALLG OR CALLS, AFTER PROBING THE EXTENT OF THE STACK
                                                ;29745
                                                ;29746  =0      ;--------------------------------;CALL SITE FOR MPUSH
                                                ;29747  CALL.7: D_Q.AND.RC[T2],                 ;STRIP MASK TO BITS 11-0
6557K     0  U 11F4, 0811,2035,0180,F910,0000,0CD8   ;29748          CALL,J/MPUSH           ;PUSH REGISTERS
                                                ;29749
                                                ;29750          ;--------------------------------;RETURN FROM MPUSH
                                                ;29751          CACHE_D[LONG],                  ;PUSH PC
6557K 7763K U 11F5, 0000,003C,0180,3270,0000,134A   ;29752          LAB_R[SP]              ; BY SP
                                                ;29753
                                                ;29754          ;--------------------------------;
6856K     0  U 134A, 0018,0000,0180,FAF0,0200,134C   ;29755  CALL.8: R[SP]&VA_LA-K[.8]       ;UPDATE SP FOR PUSH OF PC &
                                                ;29756
                                                ;29757          ;--------------------------------;
6856K     0  U 134C, 0800,003C,0180,FA68,0000,11F8   ;29758          D_R[FP]                ;READY TO PUSH FRAME POINTER
                                                ;29759
                                                ;29760  =0      ;--------------------------------;CALL SITE FOR PSHSP
                                                ;29761          CACHE_D[LONG],                  ;STORE FP,
                                                ;29762          LAB_R[SP],                      ; GET SP AGAIN
                                                ;29763          SC_K[.FFF0],                    ;-16 TO SC
6856K   21M  U 11F8, 0000,003D,6D80,3270,0084,6CD9   ;29764          CALL,J/PSHSP
                                                ;29765
                                                ;29766          ;--------------------------------;
                                                ;29767          D_R[AP],                        ;READY TO PUSH AP
6856K     0  U 11F9, 0800,003C,3DF0,2E60,0000,134D   ;29768          Q_ID[PSL]              ; AND GET PSW FOR COMBINATIO
                                                ;29769
                                                ;29770          ;--------------------------------;
                                                ;29771          CACHE_D[LONG],                  ;STORE OLD AP
                                                ;29772          Q_Q.ANDNOT.K[.1F],              ;CLEAR PSW<T,N,Z,V,C>
6856K   21M  U 134D, 0019,2024,8DC0,3270,0000,134E   ;29773          LAB_R[SP]              ;GET SP INTO LATCHES AGAIN
                                                ;29774
                                                ;29775          ;--------------------------------;
6856K     0  U 134E, 2010,0038,0180,F909,4200,1350   ;29776          PC&VA_RC[T1], FLUSH.IB ; LOAD NEW PC AND CLEAR OUT
                                                ;29777
                                                ;29778          ;--------------------------------;
                                                ;29779          D_DAL.SC,                       ;PSW TO D<31:16>
                                                ;29780          Q_RC[T2],                       ;RECOVER MASK
                                                ;29781          SC_SC+K[.3],                    ;PUT -13 IN SC
6856K     0  U 1350, 0D10,0038,0DC0,6114,0084,9351   ;29782          LOAD.IB, PC_PC+1       ;START FETCHING SUBROUTINE I
                                                ;29783
                                                ;29784          ;--------------------------------;
                                                ;29785          D_DAL.SC,                       ;MASK AND PSW IN D<31:03>
                                                ;29786          Q_PC[T4],                       ;GET LOW BITS OF OLD SP TO Q<1:0>
6856K     0  U 1351, 0D10,0038,F5C0,F920,0084,9352   ;29787          SC_SC+K[.A]            ;PUT -3 IN SC
                                                ;29788
```

# Implementing Complex Instructions




rd <= M[(rs1)] op (rs2) — *Reg-Memory-src ALU op*

M[(rd)] <= (rs1) op (rs2) — *Reg-Memory-dst ALU op*

M[(rd)] <= M[(rs1)] op M[(rs2)] — *Mem-Mem ALU op*

# Mem-Mem ALU Instructions:

***RISC-V-Controller-2***

*Mem-Mem ALU op* M[(rd)] <= M[(rs1)] op M[(rs2)]

| | | |
|---|---|---|
| $ALUMM_0$ | MA <= Reg[rs1] | next |
| $ALUMM_1$ | A <= Memory | spin |
| $ALUMM_2$ | MA <= Reg[rs2] | next |
| $ALUMM_3$ | B <= Memory | spin |
| $ALUMM_4$ | MA <=Reg[rd] | next |
| $ALUMM_5$ | Memory <= func(A,B) | spin |
| $ALUMM_6$ | | fetch |

Complex instructions usually do not require datapath modifications in a microprogrammed implementation

-- only extra space for the control program

Implementing these instructions using a hardwired controller is difficult without datapath modifications

# Performance Issues

Microprogrammed control

=> multiple cycles per instruction

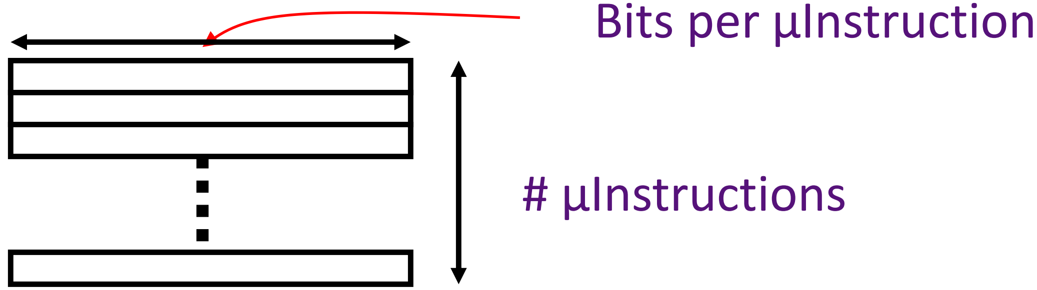
Cycle time ?

$t_C > \max(t_{reg\text{-}reg}, t_{ALU}, t_{\mu ROM})$

Suppose $10 * t_{\mu ROM} < t_{RAM}$

*Good performance, relative to a single-cycle hardwired implementation, can be achieved even with a CPI of 10*

# Horizontal vs Vertical µCode

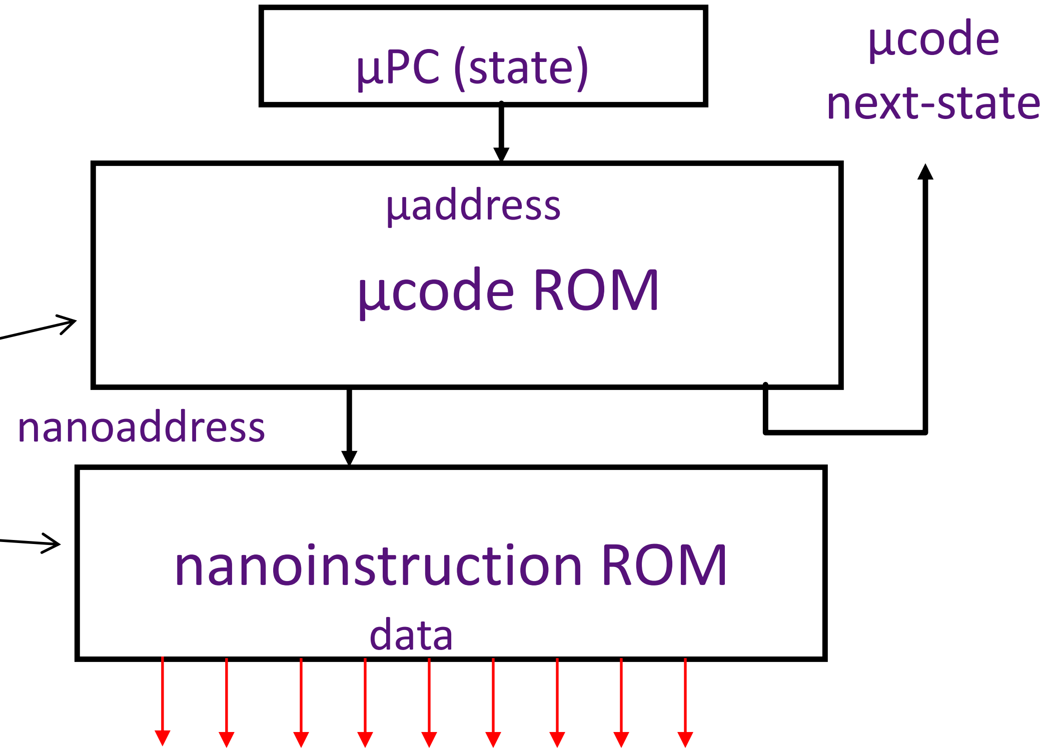



- Horizontal µcode has wider µinstructions
  - Multiple parallel operations per µinstruction
  - Fewer microcode steps per macroinstruction
  - Sparser encoding ⇒ more bits
- Vertical µcode has narrower µinstructions
  - Typically a single datapath operation per µinstruction
    - separate µinstruction for branches
  - More microcode steps per macroinstruction
  - More compact ⇒ less bits
- Nanocoding
  - Tries to combine best of horizontal and vertical µcode

# Nanocoding

Exploits recurring control signal patterns in µcode, e.g.,

$ALU_0$ A <= Reg[rs1]
...
$ALUi_0$ A <= Reg[rs1]
...




- MC68000 had 17-bit µcode containing either 10-bit µjump or 9-bit nanoinstruction pointer
  - Nanoinstructions were 68 bits wide, decoded to give 196 control signals

# Microprogramming thrived in the Seventies

- Significantly faster ROMs than DRAMs were available
- For complex instruction sets, datapath and controller were *cheaper and simpler*
- *New instructions* , e.g., floating point, could be supported without datapath modifications
- *Fixing bugs* in the controller was easier
- ISA compatibility across various models could be achieved easily and cheaply

*Except for the cheapest and fastest machines, all computers were microprogrammed*

# Writable Control Store (WCS)

- Implement control store in RAM not ROM
  - MOS SRAM memories now almost as fast as control store (core memories/DRAMs were 2-10x slower)
  - Bug-free microprograms difficult to write
- User-WCS provided as option on several minicomputers
  - Allowed users to change microcode for each processor
- User-WCS *failed*
  - Little or no programming tools support
  - Difficult to fit software into small space
  - Microcode control tailored to original ISA, less useful for others
  - Large WCS part of processor state - expensive context switches
  - Protection difficult if user can change microcode
  - Virtual memory required *restartable* microcode

# Microprogramming is far from extinct

- Played a crucial role in micros of the Eighties
  - DEC uVAX, Motorola 68K series, Intel 286/386
- Plays an assisting role in most modern micros
  - e.g., AMD Bulldozer, Intel Ivy Bridge, Intel Atom, IBM PowerPC, …
  - Most instructions executed directly, i.e., with hard-wired control
  - Infrequently-used and/or complicated instructions invoke microcode
- Patchable microcode common for post-fabrication bug fixes, e.g. Intel processors load µcode patches at bootup
  - Intel released microcode updates in 2014 and 2015

# Question of the Day

- What purpose does microcode serve today?
  - Would we have it if designing ISAs from scatch?
  - Why would we want a complex ISA?
  - Why do you think motivated CISC and RISC?